Introduction to Computer Science
Lecture 7

Nazareno Aguirre

(based on material by Guillaume Hoffmann)

Today's Topics

- Good Programming Practices
- main()'s first parameter: argc
- Color Output

Guidelines for your C programs

- Main rule:

Programs must be functionally correct, and also clear, readable and easy to understand.

- In this course and future ones (Int. Systems Programming, Digital Systems, Data Structures), we will be paying close attention to your coding style and taking it into consideration when marking/assigning grades.

- Here are a few guidelines about coding style.

Naming

- Good code should be mostly self-documenting.
- Variables names should make it clear what you are doing.
- At the moment, we are still writing short and simple programs, so typically short generic names are sufficient:
  - indexes: i, j, k,…
  - accumulators/user input: a, b, c...
- For longer and more complex programs, variable names must be more meaningful of the entities that the variables represent, avoiding very long names.

Naming Examples

Purpose of Variable	Good Names, Good Descriptors	Bad Names, Poor Descriptors
Running total of checks written to date	runningTotal, checkTotal	written, ct, checks, CHKTTL, x, x1, x2
Velocity of a bullet train	velocity, trainVelocity, velocityInMph	velt, v, tv, x, x1, x2, train
Current date	currentDate, todaysDate	cd, current, c, x, x1, x2, date
Lines per page	linesPerPage	lpp, lines, l, x, x1, x2

Optimal Name Length

- The optimal length seems to be somewhere between

  `x`

  and

  `maximumNumberOfPointsInModernOlympics`

- Names that are too short do not convey enough meaning. In small programs these are sometimes acceptable.

- Names that are too long can obscure the visual structure of the program.

Multiple-Word Variable Names

- Multiple-word variables should be formatted consistently.
- For example, "`hashtable_array_size`" or "`hashtableArraySize`" are both okay, but "`hashtable_arraySize`" is not.
- If you use "`hashtable_array_size`" in one place in a program, using "`hashtableArray`" somewhere else in the program would not be okay.
- Some developers prefer the `camelCase` naming style ("`hashtableArray`", etc.). Others the

Comments

- Comments should be present in your programs, but not excessively.
- Comments can be useful in:
  - File header: a good place to put your name and email address, and a comment describing the purpose of the file if it fits into a larger project
  - Large blocks of code: If a block of code is particularly long, a comment at the top can help the reader know what to expect as they're reading it, and let them skip it if it's not relevant.
  - Tricky bits of code: If there's no way to make some code self-evident, then it is acceptable to describe what it does with a comment.

Indentation

- Proper indentation can greatly increase the readability of code.
- Every time you open a block of code ("if" statement, "for" or "while" loop, a function, etc.), you should indent one additional level.
- You are free to use your own indent style, but you must be consistent: if you use 2 spaces as an indent in some places, you should not use 4 spaces or a tab elsewhere.
- Not sure what style to use? Use Kernighan & Ritchie style:

```
$ indent -kr myfile.c
```

Line Length

- We require program lines be **no longer than 80 characters**, so that code can be viewed without side scrolling.
  - (This is due to historical reasons, but it's still mostly the case in terminals)
  - It also helps to maintain a relatively simple program structure
- If you indent with tabs, assume tab size of 2 characters when calculating line lengths.
- To see the maximum line length of `file.c`, run:

```
$ wc -L file.c
```

No Magic Numbers

- Magic numbers are numbers in your code that have more meaning than simply their own values.
- For example: "`for(i=0;i<20;i++) x += i;`" and then "`x/20`"
- 20 is a magic number.
- Using named constants also makes your program easier to modify.
- Use `#define` to clarify the meaning of magic numbers. In the above example:

  #define TOTAL 20

  then "`for(i=0;i<TOTAL;i++) x += i;`" and then "`x/TOTAL`"

No Dead Code

- Dead code is code that is not run when your program runs.
- Sometimes because of an always true/false condition, sometimes because of an early break or return.
- Your submissions should have no dead code in it.

```
int foo(void) {
    int a = 24;
    int b = 25; // assignment to dead variable
    int c;
    c = a * 4;
    return c;
    b = 24; // unreachable code
    return 0;
}
```

No Surprises

- Code must be as straightforward as possible.
- Use good programming practices.
    - Structured programming / Good forms of iteration
- Avoid unnecessary variables.
- Avoid unnecessary features.
    - If arrays are not needed, do not use them.
- You impose more work on who reads your code when you add unnecessary things to your programs.

Fixing Bugs

When doing a programming homework or project, you will often realize that your program has bugs that need to be fixed.

Set a time limit:

- 0 minutes: `gcc -Wall`
- 1 – 10 minutes: active debugging: say out loud what the problem is, what you expect your code to do, read it step by step from the beginning.
- Speak to someone near you.
- > 10 minutes: take a break / ask for help.

Summary

- Programs can become very complex and difficult to reason about
- It is important that we favor readability in programs to favor extensibility/reuse/maintenance
  - Programs need to be read to be extended, reused, and maintained.
  - A lot more time is spent in extension/reuse/maintenance than the time spent for the initial program construction
- Programs will most likely have defects
  - Be systematic about finding and fixing bugs
  - Use tool support for debugging

main()'s first parameter: argc

- We have been using `main()` as a function with no parameters.
- Think of `main()`'s parameters as variables that tell how the program was executed from the command line.
- main() can have two specific parameters.
  - We will see the first one now: argc.
  - `argc` stands for argument count.
  - It's the number of words in the command line used when the program was executed.

Example

- Write a program that prints the value of the integer parameter argc using printf. Try:

```
$ gcc argc.c -o argc

$ ./argc

$ ./argc x

$ ./argc x x

$ ./argc x x x x x
```

Escape Sequences and the Escape Character

- An *escape sequence* is a sequence of characters that has a meaning other than the literal characters of the sequence.
    - Typically used to represent non-printable characters, and send other styling commands to the terminal.
        - \e : represents the escape character (ESC in ASCII table, value $1B)
- Escape sequences are supported by many programming language compilers and interpreters (supported by gcc and tcc, although not part of the C standard)

ANSI Escape Codes

- ANSI Escape Codes are enabled when printing the Escape Character.
- Almost all terminal emulators support them.
- Three main categories of codes:
  - text attributes: bold, underline, blink
  - foreground (text) color
  - background color

List of Attributes and Colors

Escape sequence	Text attributes
\x1b[0m	All attributes off(color at startup)
\x1b[1m	Bold on(enable foreground intensity)
\x1b[4m	Underline on
\x1b[5m	Blink on(enable background intensity)
\x1b[21m	Bold off(disable foreground intensity)
\x1b[24m	Underline off
\x1b[25m	Blink off(disable background intensity)

Sources:

https://tforgione.fr/posts/ansi-escape-codes/

https://github.com/shiena/ansicolor/

Escape sequence	Foreground colors
\x1b[30m	Black
\x1b[31m	Red
\x1b[32m	Green
\x1b[33m	Yellow
\x1b[34m	Blue
\x1b[35m	Magenta
\x1b[36m	Cyan
\x1b[37m	White
\x1b[39m	Default(foreground color at startup)
\x1b[90m	Light Gray
\x1b[91m	Light Red
\x1b[92m	Light Green
\x1b[93m	Light Yellow
\x1b[94m	Light Blue
\x1b[95m	Light Magenta
\x1b[96m	Light Cyan
\x1b[97m	Light White

Escape sequence	Background colors
\x1b[40m	Black
\x1b[41m	Red
\x1b[42m	Green
\x1b[43m	Yellow
\x1b[44m	Blue
\x1b[45m	Magenta
\x1b[46m	Cyan
\x1b[47m	White
\x1b[49m	Default(background color at startup)
\x1b[100m	Light Gray
\x1b[101m	Light Red
\x1b[102m	Light Green
\x1b[103m	Light Yellow
\x1b[104m	Light Blue
\x1b[105m	Light Magenta
\x1b[106m	Light Cyan
\x1b[107m	Light White

Example in a C program

```c
#include <stdio.h>

#define RESET   "\e[0m"
#define GREEN   "\e[102m"
#define BLUE    "\e[104m"
#define MAGENTA "\e[105m"

int main(){
    printf("%sHello %sWorld!%s\n", GREEN, MAGENTA, RESET);
    return 0;
}
```

```
$ tcc -w -run color.c
Hello World!
$
```